

SEEKER: Demand-Driven Flow-Sensitive Points-to Analysis for Java

Qiang Sun¹ Kejun Xiao² Yuting Chen² Jianjun Zhao²

¹Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China

²School of Software, Shanghai Jiao Tong University, Shanghai, China

{sun-qiang,919}@sjtu.edu.cn, {chenyt,zhao-jj}@cs.sjtu.edu.cn

Abstract

Points-to analysis is a static code analysis technique that establishes the relationships between variables of references and allocated objects. A flow-sensitive points-to analysis can achieve precise points-to relations in a program in that the flow-sensitivity information is taken into account, an iterative dataflow analysis framework is adopted, and a propagation of the points-to relations is performed through the control flows of the program, meanwhile it is resource-intensive in practice in that a number of points-to relations may be computed for all variables. In this paper, we propose SEEKER, a flow-sensitive approach to demand-driven points-to analysis of Java program, in order to query about the points-to relations of some variables of interests. In order to do this, we redefine the notion of context-free language (CFL) reachability so that it can help explore all flow-sensitive points-to relations related to the objective variables. We then design two generic algorithms for leveraging the reachability in points-to analysis.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms Analysis

Keywords points-to analysis, flow-sensitive, context-free language reachability

1. Introduction

Points-to analysis is an analysis technique which is widely used in compiler optimization, software verification and testing, and safety analysis [2, 5, 21, 24]. The goal of points-to analysis is to compute a points-to relation between variables of references and allocated objects. Flow-sensitivity is a major aspect of points-to analysis for improving the precision of the analysis [3, 4, 9]. It is mostly based on the classic iterative dataflow analysis framework [4], in which the points-to relations are computed in the control flow graph (CFG) of a program iteratively: every CFG node is associated with a transfer function, which computes the points-to

relations at two program points before and after the CFG node. Although much work [6, 12, 15] has optimized the dataflow analysis framework, a flow-sensitive points-to analysis is resource-intensive in practice in that a large number of flow-sensitive points-to relations are computed for all the variables.

In order to improve the scalability of the analysis, much recent work [16, 21, 27] focuses on demand-driven points-to analysis. A demand-driven points-to analysis only queries the points-to relations for some variables of interests, and thus reduces the computation cost through only computing the variables and their points-to relations that the objective variable depends on. One fundamental approach [21] is to formulating the points-to analysis in terms of *context-free language* (CFL) reachability [18]. The approach first abstracts a *pointer assignment graph* (PAG) [13] for a program. After that, it establishes a points-to relation $v \hookrightarrow o$, if it finds a path π in the PAG from o to v such that the word formed by concatenating the edges labels of π is in a defined CFL. However, when applied to flow-sensitive points-to analysis, the CFL reachability faces two challenges in that

- A points-to analysis usually adopts the CFL reachability to explore the points-to paths in a PAG, which is a flow-insensitive program representation not distinguishing the variables at different program points. Although we can extend the PAG by adding extra flow-sensitive information to each variable at each program point, the CFL reachability has to be redefined in order to propagate points-to relations among a number of flow-sensitive variables and explore the flow-sensitive points-to paths.
- The precision of a demand-driven points-to analysis is usually lower than that of an exhaustive flow-sensitive points-to analysis in that it omits the flow information in the program and may explore a number of infeasible points-to paths in the PAG. Meanwhile, a demand-driven points-to analysis also needs to take extra efforts to handle these redundant paths. Thus it is necessary to improve a demand-driven points-to analysis so that it not only holds the same precision as exhaustive flow-

sensitive analysis does, but also reduces the cost spent on the redundant paths.

In this paper, we propose a flow-sensitive approach, called SEEKER, to demand-driven points-to analysis for Java. We introduce the flow-sensitivity into the CFL reachability formulation in order to perform a flow-sensitive demand-driven points-to analysis. Our analysis is performed on a flow-sensitive *points-to dependence graph* (PtDG) constructed for the objective program. Each node of a PtDG is an object or a variable with program point information. Each edge of a PtDG models the points-to set propagation constraints on the variables and objects. A flow-sensitive *context-free language* (CFL) reachability is defined on the PtDG for supporting the demand-driven points-to analysis, which helps find the path of the PtDG from each variable to each object and infer whether the points-to relation holds in a flow-sensitive manner. Two generic algorithms and six implementations are also designed for providing with support in performing the analysis of different precisions.

The paper makes the following contributions:

- **Abstraction.** We present a new program representation, named the points-to dependence graph (PtDG), which abstracts both control flows and constraints on points-to sets in the program. A flow-sensitive points-to analysis that is either context-sensitive or context-insensitive points-to analysis can then be performed on the graph.
- **Definition.** We define a flow-sensitive CFL reachability on the basis of the PtDG. The CFL can be used to abstract the control flow information in the program, and the definition of reachability can provide with support in exploring all flow-sensitive points-to paths related to the objective variables as well as removing the redundant and infeasible ones.
- **Algorithm.** We design two generic algorithms for flow-sensitive points-to analysis (i.e., FSCS and FSCI), which can be implemented with different context models and alias checking algorithms. We also design six implementations of the two generic algorithms in order to take a tradeoff between the analysis cost and precision.

The remainder of the paper is organized as follows. Section 2 presents an example to illustrate how the analysis algorithm works. Section 3 presents the flow-sensitive representation of Java programs. Section 4 presents the CFL reachability formulation. Section 5 presents the details of the algorithm for flow-sensitive points-to analysis of Java programs. Section 6 discusses the related work. Section 7 concludes this paper.

2. An Illustrative Example

We next present an example to show how our analysis works. Figure 1 shows an example of Java program containing three classes, `Vector`, `Element` and `Sample`, where class

```

1 class Vector {
2     Object[] data;
3     int count;
4     Vector(){
5         Object[] t = new Object[9];
6         this.data = t;
7     }
8     void add(Object e){
9         Object[] t = this.data;
10        t[count++] = e;
11    }
12    Object get(int i){
13        Object[] t = this.data;
14        Object p = t[i];
15        return p;
16    }
17 }
18 class Element { Object f; }
19 class Sample {
20     public static void main(String[] args) {
21         Vector v = new Vector();
22         Vector w = new Vector();
23         Element a = new Element();
24         Object d = new Integer(0);
25         a.f = d;
26         v.add(a);
27         Element b = new Element();
28         w.add(b);
29         d = new String();
30         a = (Element) w.get(0);
31         a.f = d;
32         System.out.println(d);
33     }
34 }

```

Figure 1. A Sample Program.

`Vector` has a field `data` of the type `Object` array, and class `Element` has a field `f` of the type `Object`. An execution of the program starts from the method `main()` in class `Sample`, which creates two vectors `v` and `w` and accesses their elements through invoking the methods `add()` and `get()`.

In Java, an object is created by an allocation statement. Therefore we identify a set of objects $\{o_5, o_{21}, o_{22}, o_{23}, o_{24}, o_{27}, o_{29}\}$ from the program, where each subscript denotes the line number at which an object is created. For computing the points-to set of `a` after line 31, SEEKER performs the query by taking the following three steps. The first two steps are done before querying the points-to set. And the third step can then be repeated for every query.

The first step is to compute the call information for deriving all reachable methods in an analysis. The call information includes method invocations, points-to sets of receivers, and target methods. For each method invocation in form of `r.m()`, SEEKER performs a *flow-insensitive and context-insensitive* (FICI) points-to analysis [13] to compute the points-to set of the receiver `r`. SEEKER then identifies the classes of objects in the points-to set, and looks for the method declarations defined in these classes. Table 1 shows

Call Site	Receiver	Target Methods
21: Vector v = new Vector();	{o ₂₁ }	Vector.Vector()
22: Vector w = new Vector();	{o ₂₂ }	Vector.Vector()
26: v.add(a);	{o ₂₁ }	Vector.add()
28: w.add(a);	{o ₂₂ }	Vector.add()
30: a = (Element) w.get(0);	{o ₂₂ }	Vector.get()

Table 1. Target Methods for the Call Sites.

the call information in the program, including all the call sites, the corresponding receiver points-to sets and target methods.

The second step is to build a CFG and a PtDG for the objective program. Figure 2 shows the CFG of the sample program built by SEEKER. Each node is identified by a unique name; each simple statement or a predicate of a condition/loop statement corresponds to a node in the CFG, if it can change the points-to relations in the program; each method contains two auxiliary nodes (i.e., **Enter** and **Exit**); each method invocation statement corresponds to **Call** and **Return** nodes, e.g., (21⁻, 21'), (22⁻, 22'), (26, 26'), (28, 28') and (30, 30') correspond to the call sites at lines 21, 22, 26, 28 and 30, respectively. SEEKER reduces the size of the CFG by excluding all statements which cannot modify the points-to relations in the program, such as `print()` statements and the assignments of variables of primitive types.

Based on the CFG, SEEKER constructs a PtDG shown in Figure 3. Each node of PtDG is an object or a variable with a program point in form of $x@n_o$ or $x@n_\bullet$. For example, $v@21^-$ and $w@22_\bullet$ represent the state of the variable v before the execution of 21⁻ and that of w after the execution of 22, respectively. An edge in a PtDG can be (1) a copy edge if it does not have any label, which denotes that the points-to set of source node is propagated to that of target node; (2) a transitive edge of a label $*$, which denotes that the source node reaches the target node through a sequence of copy edges; (3) an interprocedural edge (i.e., the dot-dash edge), which denotes a passing of a parameter or an assignment using a return value; and (4) the edge labeled with *new*, $n\#ld[f]$, or $n\#st[f]$, which corresponds to an allocation, load, or store statement. Note that in a load or store edge, the label employs '#' to separate the corresponding CFG node and the access type of the edge.

The third step is to find the flows-to paths with respect to the variable a at line 31. A flows-to path is composed of a sequence of adjacent edges and nodes that is CFL reachable to the variable of interest. A node in form of $(State, Context)$ contains a state of certain variable at specific program point and a call string that reaches the variable. Note that *Context* is modeled by a sequence of line numbers of call sites. A CFL reachability [18, 19] is an extension of graph reachability problem, which allows for identifying the valid path

through taking into account the edge labels. A flows-to path to the variable a at line 31 (i.e., $(a@31_\bullet, \epsilon)$) is

$$\begin{aligned} & \dots \rightarrow (p@14_\bullet, 30) \rightarrow (p@15_o, 30) \rightarrow (ret_{get}@15_\bullet, 30) \\ & \rightarrow (ret_{get}@16_o, 30) \rightarrow (a@30'_\bullet, \epsilon) \\ & \rightarrow (a@31_o, \epsilon) \rightarrow (a@31_\bullet, \epsilon) \end{aligned}$$

Note that the line numbers are not corresponded to the numbers of CFG nodes. A computation of the points-to set of $(a@31_\bullet, \epsilon)$ is then transformed to a computation of the points-to set of $(p@14_\bullet, 30)$.

When the algorithm handles a load edge, it collects all the store edges which potentially reach the load edge. Then the algorithm uses the alias checking to identify the matched load/store edges which enable the propagation of points-to sets.

For the load edge

$$(p@14_\bullet, 30) \xleftarrow{14\#ld[arr]} (t_{get}@14_o, 30),$$

the algorithm collects two store edges in different contexts

$$\begin{aligned} (t_{add}@10_\bullet, 26) & \xleftarrow{10\#st[arr]} (e_{add}@10_o, 26), \\ (t_{add}@10_\bullet, 28) & \xleftarrow{10\#st[arr]} (e_{add}@10_o, 28), \end{aligned}$$

where *arr* represents a special field to model the accessing of the elements of an object array. Note that the node 10 can reach the node 14 in the CFG (see Figure 2). After that, it computes the following results for alias checking

$$\begin{aligned} PointsTo((t_{get}@14_o, 30)) &= \{(o_5, 22)\}, \\ PointsTo((t_{add}@10_\bullet, 28)) &= \{(o_5, 22)\}, \\ PointsTo((t_{add}@10_\bullet, 26)) &= \{(o_5, 21)\}. \end{aligned}$$

An alias check can help find that

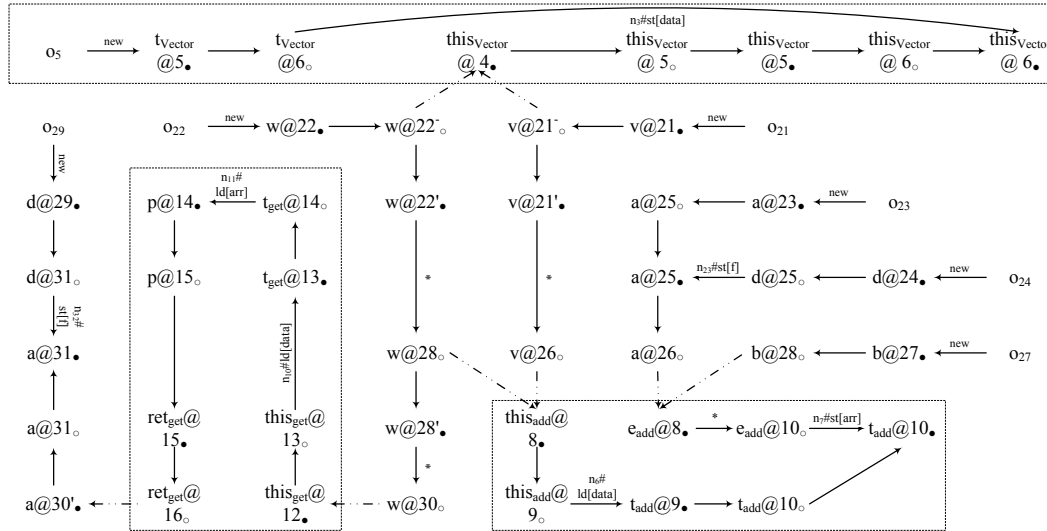
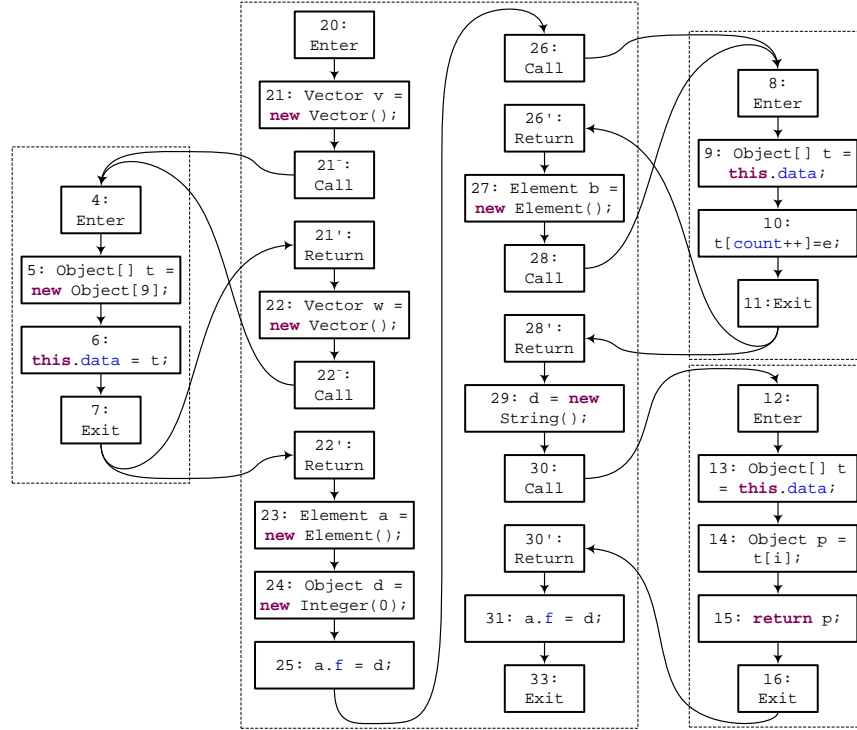
$$((t_{get}@14_o, 30), (t_{add}@10_\bullet, 26))$$

is not an alias pair because the disjunction of their points-to sets is empty, but

$$((t_{get}@14_o, 30), (t_{add}@10_\bullet, 28))$$

is because the points-to sets of the variables in the pair contain $(o_5, 22)$. A detailed explanation of the algorithm for alias checking is given in Section 5.2. Therefore, the points-to set of $(e_{add}@10_o, 28)$ is propagated to that of $(p@14_\bullet, 30)$. The points-to set of $(e_{add}@10_o, 28)$ is determined by the path

$$\begin{aligned} (o_{27}, \epsilon) & \xrightarrow{new} (b@27_\bullet, \epsilon) \rightarrow (b@28_o, \epsilon) \\ & \rightarrow (e_{add}@8_\bullet, 28) \xrightarrow{*} (e_{add}@10_o, 28). \end{aligned}$$



An FSCS algorithm given in Section 5.1 can infer that there exists only one path from $\{(o_{27}, \epsilon)\}$ to $(a@31_{\bullet}, \epsilon)$

Thus we can conclude that the points-to set of $(a@31_\bullet, \epsilon)$ is $\{(o_{27}, \epsilon)\}$.

$$\begin{aligned}
(o_{27}, \epsilon) &\xrightarrow{new} (b@27_{\bullet}, \epsilon) \rightarrow (b@28_{\circ}, \epsilon) \rightarrow (e_{add}@8_{\bullet}, 28) \\
&\xrightarrow{*} (e_{add}@10_{\circ}, 28) \xrightarrow{10\sharp st[arr]} (t_{add}@10_{\bullet}, 28) \xrightarrow{alias} \\
&(t_{get}@14_{\circ}, 30) \xrightarrow{14\sharp td[arr]} (p@14_{\bullet}, 30) \rightarrow (p@15_{\circ}, 30) \\
&\rightarrow (ret_{get}@15_{\bullet}, 30) \rightarrow (ret_{get}@16_{\circ}, 30) \rightarrow (a@30'_{\bullet}, \epsilon) \\
&\rightarrow (a@31_{\circ}, \epsilon) \rightarrow (a@31_{\bullet}, \epsilon)
\end{aligned}$$

3. Program Representation

3.1 An Abstraction of Java Programs

A program in Java is composed of a set of classes *CLASS*. Next we define some program notions, which will be further used in our analysis:

- A set V containing all local variables, formal parameters in the methods, and static fields in the classes.
- A set of class fields F , each of which, say f , can be accessed using the form $v.f$, $v \in V$.
- A set of methods M , each of which has a formal parameter list $fp_1, \dots, fp_n \in V$ and a return value $ret \in V$. Note that a receiver variable is passed to *this* parameter $fp_0 \in V$ implicitly.
- A set $Stmt$ containing all statements in the program. A statement can be:
 - A copy statement: $v_1 = v_2$,
 - An allocation statement: $v = \text{new } C()$,
 - A load statement: $v_1 = v_2.f$,
 - A store statement: $v_1.f = v_2$,
 - A return statement: $\text{return } v$,
 - A method invocation statement $rt = v_0.m(v_1, \dots, v_k)$.
- A set O containing all names of objects created by the object allocation statements.

3.2 Call Information Generation

A points-to relation is defined between a reference $r \in R$ and its allocation sites $o \in O$, where $R = V \cup O \times F$ (i.e., a reference can be a variable $v \in V$ or an object field $o.f \in O \times F$). A points-to graph is a graphical representation of a set of points-to relations. The points-to set of a variable v is defined by

$$\{o | v \hookrightarrow o, o \in O\}.$$

In this study we adopt a *flow-insensitive and context-insensitive* (FICI) points-to analysis algorithm in Spark [13] to calculate the call information in the objective program.

For each call site, the target methods are deduced through identifying the types of the objects in the points-to set of the receiver. The method call information is represented by using a mapping

$$\tau \in (Stmt \rightarrow \mathcal{P}(M))$$

where $\mathcal{P}(M)$ represents the power set of M .

3.3 Control Flow Graph Construction

A *control flow graph* (CFG) [8] is constructed for analyzing all reachable methods and taking a flow-sensitive points-to analysis. A CFG is composed of a set of nodes and edges. A node $n \in N$ in a CFG can be

- a statement node s representing a statement in $Stmt$,

- an enter node $en \in ENTER$ denoting the unique entry to a method,
- an exit node $ex \in EXIT$ denoting the unique exit from a method,
- a call node $cn \in CALL$ modeling the location before a method invocation,
- a return node $rn \in RETURN$ modeling the location after a method invocation.

An edge $(n_1 \rightarrow n_2) \in E$ denotes n_2 is a successive execution node of n_1 . It can either be an interprocedural edge belonging to $InterE \subseteq (CALL \times ENTER) \cup (EXIT \times RETURN)$, or an intraprocedural edge belonging to $IntraE \triangleq E \setminus InterE$. Specially, an interprocedural edge denotes a relation between an invocation statement and a target method, which benefits the context-sensitive analysis in providing with the method invocation information in form of edge labels.

The CFG construction algorithm first generates the CFG for each method, and then generates two nodes $call^i$ and $return^i$ for each invocation statement node i . After that, it generates a set of edges

$$\begin{aligned} set = & \{call^i \rightarrow enter^m | m \in \tau(i)\} \cup \\ & \{exit^m \rightarrow return^i | m \in \tau(i)\} \cup \\ & \{n \rightarrow call^i | (n \rightarrow i) \in E\} \cup \\ & \{return^i \rightarrow n | (i \rightarrow n) \in E\} \end{aligned}$$

where $\tau(i)$ represents a target method set of i . Finally, all the edges related to the invocation statements are removed because they are also modeled by the interprocedural edges.

3.4 Points-to Dependence Graph Construction

SEEKER enables strong updates [12] through distinguishing the variables of different program points (i.e., before or after a CFG node). It adds the control flow information into the constraints, and then computes a possibly different result for each program point.

To this end, we present a *points-to dependence graph* (PtDG) abstracting both the control flows and constraints on the points-to sets in the program. We define a flow-sensitive variable set $V_{FS} \subseteq V \times N \times \{before, after\}$. A node in a PtDG can be a flow-sensitive variable node in $v' \in V_{FS}$, or an object node $o \in O$. We use the notations $v@n_o$ and $v@n_\bullet$ to denote the variable v before and after the node n , respectively and use $v@*$ to denote the variable of all program point. An edge in a PtDG can be an **allocation**, **copy**, **load** or **store** edge.

A PtDG can be constructed according to the rules in Figures 4 and 5. Figure 4 describes the rules for generating all the intraprocedural PtDG edges. The constraints [ALLO-CATE], [COPY1], [LOAD], [STORE], and [RETURN] are similar to those in the flow-insensitive analysis, except they now relate a points-to set before each statement with an-

$n : v = \text{new } C()$	$o \xrightarrow{\text{new}} v@n_\bullet$	[ALLOCATE]
$n : v_1 = v_2$	$v_2@n_o \xrightarrow{\text{assign}} v_1@n_\bullet$	[COPY1]
$n : v = sf$	$sf@* \xrightarrow{\text{assignglobal}} v@n_\bullet$	[COPY2]
$n : sf = v$	$v@n_o \xrightarrow{\text{assignglobal}} sf@*$	[COPY3]
$n : sf_1 = sf_2$	$sf_2@* \xrightarrow{\text{assignglobal}} sf_1@*$	[COPY4]
$n : v_1 = v_2.f$	$v_2@n_o \xrightarrow{n\#ld[f]} v_1@n_\bullet$	[LOAD]
$n : sf = v.f$	$v@n_o \xrightarrow{n\#ld[f]} sf@*$	[SLOAD]
$n : v_1.f = v_2$	$v_2@n_o \xrightarrow{n\#st[f]} v_1@n_\bullet$	[TORE]
$n : v.f = sf$	$sf@* \xrightarrow{n\#st[f]} v@n_\bullet$	[SSTORE]
$n : \text{return } v$	$v@n_o \xrightarrow{\text{assign}} \text{ret}^m@n_\bullet$	[RETURN]
$(n_1 \rightarrow n_2) \in \text{IntraE}$	$\forall v \in V_m. \quad v@n_\bullet \xrightarrow{\text{assign}} v@n_o$	[FLOW]
$n \in \text{Stmt}$	$\forall v \in V_m \setminus \text{kill}(n). \quad v@n_o \xrightarrow{\text{assign}} v@n_\bullet$	[PRESERVE]

Figure 4. Rules for Generating PtDG Intraprocedural Edges.

other set after that statement. The rules [COPY2], [COPY3] and [COPY4] model the points-to sets with respect to propagation of the static fields. A static field may occur at any program point. Therefore, we do not associate any static field with a specific program point due to the cost reason. The [FLOW] constraint models the effect of control flow: whenever n_2 follows n_1 in the control flow graph, the points-to sets before n_2 contain everything contained in the points-to sets after n_1 . The [PRESERVE] constraint models a statement not affecting the points-to sets of some variables: if a variable v is not killed by the statement n , the points-to set of $v@n_\bullet$ contains all the targets in the points-to set of $v@n_o$. Here the kill set is defined by

$$\text{kill}(n : v = \dots) \triangleq \{v\},$$

where $\text{kill}(n : v = \dots)$ contains all variables killed by an allocation, copy, or load statement

Figure 5 defines a rule [INVOKE] for generating three kinds of interprocedural PtDG edges:

- from the nodes of actual parameters to those of formal parameters,
- from the node of return value to that of the left-hand variable at the call site i , and
- from the nodes of variables before i to those after i , whose points-to sets are preserved.

Flow-sensitive algorithms can be combined with some flow-insensitive algorithms (e.g., those in [21, 22]) in that a PtDG for flow-sensitive analysis can be transformed to a *pointer assignment graph* [13] (PAG) on which a flow-insensitive analysis can be performed. We define an equivalent relation \mathfrak{R} on the flow-sensitive variable nodes. For $a, b \in V_{FS}$, we have

$$a \mathfrak{R} b \text{ iff } f(a) = f(b),$$

where the function f is

$$f : V_{FS} \rightarrow V, f(v@...) = v.$$

An \mathfrak{R} -equivalent class is defined by $\mathfrak{R}_v = \{a | f(a) = v, a \in V_{FS}\}$, $v \in V$. Therefore, we can transform a PtDG to a PAG through merging all the \mathfrak{R} -equivalent flow-sensitive variable nodes and eliminating the self-loops.

4. Flow-Sensitive Points-to Analysis via CFL Reachability

SEEKER is a demand-driven approach organized by answering queries for the points-to set of a specific variable. An important characteristic of the approach is to only perform calculations contributing to answering the user-defined queries, and eliminate unnecessary calculations. Thus SEEKER achieves significant time and space savings.

For a given points-to set query, SEEKER follows the notion of CFL reachability [18, 19] to generate and answer some questions related to the given query. For example, in order to compute the points-to set of $(a@31_\bullet, \epsilon)$, SEEKER raises and answers the following questions:

- What may $(p@14_\bullet, 30)$ point to?
- Does the alias relation between $(t_{get}@14_o, 30)$ and $(t_{add}@10_\bullet, 26)$ hold?
- Does the alias relation between $(t_{get}@14_o, 30)$ and $(t_{add}@10_\bullet, 28)$ hold?
- What may $(t_{add}@10_\bullet, 28)$ point to?

CFL reachability is an extension of graph reachability problem, which allows the valid paths to be identified. Let G be a directed graph, the edges of which are labeled by symbols of an alphabet Σ . Then, let L be a CFL over Σ . Each path π in G is mapped to a string $w(\pi)$ in Σ^* established by concatenating the labels of edges in π in order. A node u is L -reachable from a node v if there exists a path π from v to u , called an L -path, such that $w(\pi) \in L$. Let an edge be $x \xrightarrow{l} y$ and its inverse edge be $y \xrightarrow{\bar{l}} x$. The inverse path of a path $\pi = e_1 \dots e_k$ is $\bar{\pi} = \bar{e}_k \dots \bar{e}_1$.

In order to introduce the flow-sensitivity into the CFL reachability formulation, we need to study the access order of the instance fields. We first define the relation \dagger between two CFG nodes of field-store statements. Given two nodes of a CFG

$$n_1 : v_1.f = a \text{ and } n_2 : v_2.f = b,$$

$n_2 \dagger n_1$ iff

- there is a path from n_1 to n_2 in the CFG, and
- $\text{PointsTo}(v_1) = \text{PointsTo}(v_2) = \{o\}$,

where PointsTo is a mapping which can be computed on the fly, and both the points-to set of v_1 and v_2 are singleton sets.

$$\begin{array}{l}
(call^i, enter^m, exit^m, return^i) \quad \forall 0 \leq j \leq k. v_j @ call_o^i \xrightarrow{CallEnter_i} fp_j^m @ enter_o^m, ret^m @ exit_o^m \xrightarrow{ExitReturn_i} rt @ return_o^i \quad [\text{INVOKE}] \\
i : rt = v_0.m(v_1, \dots, v_k), m \in \tau(i) \quad \forall v \in V_m \setminus \{rt\}. v @ call_o^i \xrightarrow{assign} v @ return_o^i
\end{array}$$

Figure 5. Rules for Generating PtDG Interprocedural Edges.

After that, we introduce a relation $\triangleright \subseteq S \times L$, where S and L represent the control flow node sets of field store and load statements, respectively. Given two nodes of a CFG

$$n_s : v_1.f = r \text{ and } n_l : l = v_2.f,$$

$n_s \triangleright n_l$ iff

- there is a path π from n_s to n_l in the CFG, and
- $\neg \exists (n : v.f = r') \in \pi. n \dagger n_s$.

Based on the relation \triangleright , we define the CFL formulation for two flow-sensitive points-to analyses, which are context-insensitive and context-sensitive versions. We first define a CFL L_F for a context-insensitive version. The symbol set of L_F is

$$\begin{aligned}
\Sigma_F \triangleq \{ & assign, new, n\sharp ld[f], n\sharp st[f], \\
& \overline{assign}, \overline{new}, \overline{n\sharp ld[f]}, \overline{n\sharp st[f]} \}.
\end{aligned}$$

The grammar of L_F is

$$\begin{aligned}
flowsTo &\rightarrow new(assign[n_1\sharp st[f] \text{ alias } n_2\sharp ld[f]])^* \\
alias &\rightarrow pointsTo flowsTo \\
pointsTo &\rightarrow (\overline{assign}[\overline{n_2\sharp ld[f]} \text{ alias } \overline{n_1\sharp st[f]}])^* \overline{new}
\end{aligned}$$

where n_1 and n_2 satisfy $n_1 \triangleright n_2$.

In the context-insensitive version, a variable v points to an object o iff there exists a path π from o to v in the PtDG G and $w_F(\pi) \in L_F$, where w_F is a function to translate π into a string by replacing the labels **CallEnter** and **ExitReturn** with **assign**. A definition like that improves the original definition in [22] in that it takes the order between the field accesses into account, and thus during an analysis we can eliminate the redundant points-to relations introduced by the infeasible paths of the CFG.

Then, we define a language L_C to perform context-sensitivity check for the context-sensitive version, which requires edges labeled with **CallEnter** and **ExitReturn** to be matched to reduce the infeasible points-to paths. The symbol set Σ_I of L_C is

$$\begin{aligned}
\Sigma_I \triangleq \{ & CallEnter_k, \overline{ExitReturn_k}, \\
& \overline{ExitReturn_k}, \overline{CallEnter_k} \},
\end{aligned}$$

where k represents any call site in the program.

Since it is difficult to define L_C directly, we define the complementary language L_C^- of L_C , whose symbol set is

also Σ_I . The grammar of L_C^- is

$$\begin{aligned}
C^- &\rightarrow \Sigma_I^* (i \ B)_j \Sigma_I^* \\
B &\rightarrow (i \ B)_i \mid B \ B \mid \epsilon \\
(i \rightarrow CallEnter_i \mid \overline{ExitReturn_i} \\
&)_i \rightarrow ExitReturn_i \mid \overline{CallEnter_i}
\end{aligned}$$

where i is not equal to j . If the context model contains finite contexts, L_C^- is regular for the reason that the number of terms derived from B is finite. Since the regular languages are close under the complement operation, we have a regular language $L_C = \Sigma_I^* \setminus L_C^-$, which will be applied to a finite context model.

In a PtDG G , a path π is context-sensitive valid if $w_C(\pi) \in L_C$, where w_C is a function to translate π into a string which contains the symbols in the symbol set Σ_I , but replaces the other edge labels with ϵ . A context-sensitivity check can be performed along with the identification of an L_F -path.

In the context-sensitive analysis version, a variable v points to an object o iff there exists a path π from o to v in the PtDG G and $w_F(\pi) \in L_F \wedge w_C(\pi) \in L_C$.

5. Algorithms of Flow-Sensitive Points-to Analysis

5.1 Generic Flow-Sensitive Points-to Analysis Algorithms

SEEKER performs a demand-driven points-to analysis by adopting a *flow-sensitive and context-sensitive* (FSCS) or a *flow-sensitive and context-insensitive* (FSCI) algorithm. In the following, we describe two generic flow-sensitive algorithms based on CFL reachability in order to derive different implementations.

5.1.1 FSCS Algorithm.

Algorithm 1 shows a generic FSCS algorithm. Given a variable v and a call stack c , the procedure call *FSCS_PointsTo* (v, c, \emptyset) returns the points-to set of v in context c , containing a set of pairs in form of $(object, context)$. The algorithm traverses the PtDG for looking for all incoming L_F -paths to v and filters the paths which cannot pass the L_C check. A global data structure *cache* is adopted to store the points-to set for each pair (v, c) , which ensures the points-to set of (v, c) is computed once in the analysis.

For an allocation edge $v \xleftarrow{new} o$, the FSCS algorithm adds a heap-sensitive object (o, c) into the points-to set of (v, c) (see line 10). Since the global variables are context-insensitive, the call stack c needs to be cleared if **assign-global** edges have been visited, and thus the sequence of

Algorithm 1: Algorithm for FSCS Points-to Analysis

```
input : A flow-sensitive variable  $v$ , a stack  $c$  and a set  $visited$ 
output: A points-to set  $pts$ 
1 FSCS.PointsTo( $v, c, visited$ )
2 begin
3   if  $visited.contains((v, c))$  then
4     return  $\emptyset$ 
5    $visited.put((v, c));$ 
6   if  $cache.containsKey((v, c))$  then
7     return  $cache.get((v, c))$ 
8    $pts \leftarrow \emptyset;$ 
9   for each edge  $v \xleftarrow{new} o$  do
10     $pts \leftarrow pts \cup \{(o, c)\}$ 
11   for each edge  $v \xleftarrow{assign} x$  do
12     $pts \leftarrow pts \cup FSCS.PointsTo(x, c, visited)$ 
13   for each edge  $v \xleftarrow{assignglobal} x$  do
14     $pts \leftarrow pts \cup FSCS.PointsTo(x, \epsilon, visited)$ 
15   for each edge  $v \xleftarrow{ExitReturn_i} x$  do
16     $pts \leftarrow pts \cup FSCS.PointsTo(x, c.BDPush(i), visited)$ 
17   for each edge  $v \xleftarrow{CallEnter_i} x$  do
18     if  $c.peek() = i \vee c = \epsilon$  then
19        $pts \leftarrow pts \cup FSCS.PointsTo(x, c.pop(), visited)$ 
20   for each edge  $v \xleftarrow{n_1 \#td[f]} u$  do
21     for each edge  $q \xleftarrow{n_2 \#st[f]} p, n_2 \triangleright n_1$  do
22       if  $\neg((u, c), q) \in CAlias$  then
23          $CAlias \leftarrow CAlias \cup \{(u, c), q\};$ 
24       for each  $c' \in cset$  do
25         if  $AliasCondition$  then
26            $pts \leftarrow pts \cup FSCS.PointsTo(p, c', visited)$ 
27        $CAlias \leftarrow CAlias \setminus \{(u, c), q\};$ 
28    $cache.put((v, c), pts);$ 
29   return  $pts$ 
```

calls and returns between two accessing of any global variable can be omitted on the edges (see lines 13 and 14). An incoming $ExitReturn_i$ edge requires the call site i to be pushed onto the call stack (line 16) and an incoming $CallEnter_i$ edge requires the top element to be popped off (see line 19). Note that the pushing stack algorithm (i.e., $BDPush()$) can be specialized for different context models. In addition, the check for $c = \epsilon$ at line 18 allows for partially balanced call parentheses.

In the algorithm, lines 20 – 24 are used to compute the points-to sets when the **load** and **store** edges are visited. Here $AliasCondition$ checks whether (u, c) and (q, c') are aliases or not. This check can be specialized by either a definition of *alias* (see Section 4) or other CFL-reachability algorithms (e.g., flow-insensitive and context-insensitive algorithms). If (u, c) and (q, c') are aliases, the points-to set of (q, c') is propagated to (v, c) .

We use a set $CAlias$ to record the potential alia pairs, which are currently processed. Furthermore, we use the $visited$ set to include all pairs in query (see lines 3 – 5). If a new query, say (v, c) , is in $visited$, FSCS stops the query because the query is on a cyclic path, which cannot

Algorithm 2: Algorithm for FSCI Points-to Analysis

```
input : A flow-sensitive variable  $v$  and a set  $visited$ 
output: A points-to set  $pts$ 
1 FSCI.PointsTo( $v, visited$ )
2 begin
3   if  $visited.contains(v)$  then
4     return  $\emptyset$ 
5    $visited.put(v);$ 
6   if  $cache.containsKey(v)$  then
7     return  $cache.get(v)$ 
8    $pts \leftarrow \emptyset;$ 
9   for each edge  $v \xleftarrow{new} o$  do
10     $pts \leftarrow pts \cup \{o\}$ 
11   for each edge  $v \xleftarrow{assign/assignglobal} x$  do
12     $pts \leftarrow pts \cup FSCI.PointsTo(x, visited)$ 
13   for each edge  $v \xleftarrow{n_1 \#td[f]} u$  do
14     for each edge  $q \xleftarrow{n_2 \#st[f]} p, n_2 \triangleright n_1$  do
15       if  $\neg(u, q) \in CAlias$  then
16          $CAlias \leftarrow CAlias \cup \{(u, q)\};$ 
17       if  $AliasCondition$  then
18          $pts \leftarrow pts \cup FSCI.PointsTo(p, visited)$ 
19        $CAlias \leftarrow CAlias \setminus \{(u, q)\};$ 
20    $cache.put(v, pts);$ 
21   return  $pts$ 
```

introduce any new points-to relations for the variables in the path.

5.1.2 FSCI Algorithm.

Algorithm 2 shows a generic FSCI algorithm. Given a variable v , the procedure call $FSCI.PointsTo(v, \emptyset)$ traverses edges of the points-to dependence graph to find the points-to path, and returns the points-to set of v . A cache $cache$ is used to preserve the computation results.

The FSCI algorithm is simpler than the FSCS one because it does not take into account the context information. It also does not distinguish the edges $v \xleftarrow{assign} x$ and $v \xleftarrow{assignglobal} x$ (see line 12). The structure of $visited$ and $CAlias$ do not keep the context information. $AliasCondition$ is used to check the aliases without distinguishing the contexts of variables.

5.2 Implementations of the FSCS/FSCI Algorithms

The FSCS/FSCI algorithms are generic because the context model and alias checking condition are not defined. Here we design six implementations of the FSCS/FSCI algorithms (i.e., **FullFSCS**, **FullFSCI**, **C1**, **C2**, **C3**, and **C4**) in order to resolve the tradeoff between the analysis cost and precision. In addition, we explore three flow-insensitive algorithms for points-to analysis, two of which (i.e., **FullFICS** and **FullFICI**) have been developed by Sridharan et al. [21, 22] based on PAG. FullFICS corresponds to the algorithm *Full* algorithm in [21], which is a flow-insensitive and context-sensitive algorithm. FullFICI corresponds to the algorithm *FullFS* (i.e., full field-sensitive algorithm) in [22], which is a flow-insensitive and context-insensitive algorithm.

Algorithm 3: Algorithm for Bounded Depth Pushing

```
input : A call site  $i$ 
output: A new call stack  $r$ 
1 BDPush( $i$ )
2 begin
  // this is a call stack.
3    $r \leftarrow \text{this}; t \leftarrow \epsilon;$ 
4   while ! $r.\text{isEmpty}()$  do
5     if  $r.\text{peek}() = i$  then
6       Break
7     else
8        $t.\text{push}(r.\text{pop}())$ 
9    $r \leftarrow \epsilon;$ 
10  while ! $t.\text{isEmpty}()$  do
11     $r.\text{push}(t.\text{pop}())$ 
12   $r.\text{push}(i);$ 
13  return  $r$ 
```

5.2.1 FullFSCS Algorithm.

FullFSCS is an implementation of the FSCS algorithm, which requires all user-queries as well as the intermediate queries using the FSCS algorithm. The FullFSCS algorithm addresses two important issues.

The first issue is that the objective program may have recursive method calls which introduce a context of infinite length in an FSCS analysis. Generating a context of infinite length leads to nontermination (see line 16 of Algorithm 1). We require the context of finite length stored in the stack, which is shown in Algorithm 3. Specially, when a call site i needs to be pushed into c , the algorithm searches i in c supposing i exists at the *index* position, and discards all elements from stack bottom to *index* (see lines 4-11). After that i is pushed into c . Algorithm 3 generates a finite number of contexts bounded by the number of acyclic call graph paths, because it excludes the contexts of loops. A similar technique is also used in cloning-based context-sensitive points-to analysis [26].

The second issue is to perform the alias checking of a source variable of a load edge and a target variable of a store edge with respect to some contexts. In FullFSCS, the context of the source variable needs to be the same as that of the target variable. Furthermore, an alias checking can be decomposed into two points-to set query problems, but the contexts of the target variable of the store edge can be inferred through traversing either (1) along the flows-to paths [22] or (2) along the points-to paths. Since the first approach computes an exhaustive alias set containing a great number of context-sensitive variables that are redundant for the alias checking, we adopt the second approach in FullFSCS for the sake of cost. Here we collect all the valid contexts and choose those satisfying an alias checking condition, which is explained next.

We define $cset$ to store all the valid contexts for the flow-sensitive variable q on line 22 of Algorithm 1

$$cset = \text{FINDCONTEXTS}(m, \epsilon, \emptyset),$$

Algorithm 4: Algorithm for Seeking Contexts

```
input : A method  $m$ , a context  $c$ , a set  $visited$ 
output: A context set  $cset$ 
1 FINDCONTEXTS( $m, c, visited$ )
2 begin
3    $cset \leftarrow cset \cup \{c\};$ 
4   if  $visited.\text{contains}(m)$  then
5     return  $cset$ 
6    $visited \leftarrow visited \cup \{m\};$ 
7   for each  $i, m \in \tau(i)$  do
8      $r \leftarrow c; t \leftarrow \emptyset;$ 
9     while ! $r.\text{isEmpty}()$  do
10        $t.\text{push}(r.\text{pop}())$ 
11      $r.\text{push}(i);$ 
12     while ! $t.\text{isEmpty}()$  do
13        $r.\text{push}(t.\text{pop}())$ 
14     if the call site  $i$  occurring in the method  $m'$  then
15        $cset \leftarrow cset \cup \text{FINDCONTEXTS}(m', r, visited)$ 
16  return  $cset$ 
```

where m is the method in the scope of which q is used. *FINDCONTEXTS* follows a context model consistent with that of *BDPush()* and generates the contexts for looking up the call information (see Algorithm 4).

The target variable of a store edge in different contexts may point to one object. We thus discard the contexts that may decrease the precision of analysis. We define the partial order \prec for the context set, $c_1 \prec c_2$ if c_1 is a topmost subsequence of c_2 . A context c_2 allows for more precision than c_1 on $q \hookrightarrow (o, c_o)$ if

$$(q, c_1) \hookrightarrow (o, c_o) \wedge (q, c_2) \hookrightarrow (o, c_o) \wedge c_1 \prec c_2$$

For example, give the store statement at line 10 of the program in Figure 1, we have

$$(t_{add}@10_\bullet, 28) \hookrightarrow (o_5, 22), (t_{add}@10_\bullet, \epsilon) \hookrightarrow (o_5, 22).$$

On $t_{add}@10_\bullet \hookrightarrow (o_5, 22)$, the context $\langle 28 \rangle$ allows for more precision than ϵ does.

Therefore, an alias checking condition *AliasCondition₁* is defined

$$\begin{aligned} \exists (o, c_o). (o, c_o) \in (\text{FSCS_PointsTo}(u, c, \emptyset) \cap \\ \text{FSCS_PointsTo}(q, c', \emptyset)) \wedge \\ (\neg \exists c''. c' \prec c'' \wedge (o, c_o) \in \text{FSCS_PointsTo}(q, c'', \emptyset)) \end{aligned}$$

which denotes that (u, c) and (q, c') are aliases and c' is one of the most precise contexts of q on $q \hookrightarrow (o, c_o)$.

5.2.2 FullFSCI Algorithm.

FullFSCI is an implementation of the generic FSCI algorithm. It requires all user-queries as well as the intermediate queries to be responded using the FSCI algorithm. FullFSCI defines *AliasCondition₂* in the FSCI algorithm by

$$\text{FSCI_PointsTo}(u, \emptyset) \cap \text{FSCI_PointsTo}(q, \emptyset) \neq \emptyset$$

which denotes that u and q are aliases.

5.2.3 C1-C4 Algorithms.

An alias checking algorithm can be independent from the points-to analysis algorithms for intermediate queries. We developed four composition algorithms (i.e., C1, C2, C3, and C4). C1-C3 implement Algorithm 1 through adopting different alias checking conditions.

- C1: $cset = \{\epsilon\}$. The alias checking condition *AliasCondition*₃ is

$$u = u' @_{n_1 \circ} \wedge q = q' @_{n_2 \bullet} \wedge \\ FullFICI(u', \emptyset) \cap FullFICI(q', \emptyset) \neq \emptyset$$

which denotes that u and q are aliases ignoring the program points information.

- C2: $cset = \{\epsilon\}$. *AliasCondition*₄ is

$$FullFSCI(u, \emptyset) \cap FullFSCI(q, \emptyset) \neq \emptyset$$

which denotes that u and q are aliases.

- C3: $cset$ is computed by the algorithm `FINDCONTEXTS` (Algorithm 4). *AliasCondition*₅ is

$$u = u' @_{n_1 \circ} \wedge q = q' @_{n_2 \bullet} \wedge \exists (o, c). ((o, c) \in \\ (FullFICS(u', c, \emptyset) \cap FullFICS(q', c', \emptyset)) \wedge \\ (\neg \exists c''. c' \prec c'' \wedge (o, c) \in FullFICS(q', c'', \emptyset)))$$

which denotes that (u', c) and (q', c') are aliases and c' is one of the most precise contexts of q' on $q' \hookrightarrow (o, c_o)$.

C4 implements the FSCI algorithm through adopting the alias checking condition *AliasCondition*₃.

6. Related Work

In this section, we discuss some work related to the exhaustive and the demand-driven points-to analyses.

Exhaustive Points-to Analysis. There exist two classical algorithms for points-to analysis. Andersen's algorithm [1] is a subset-based analysis with cubic worst-case complexity. It produces more precise results, which deals with the pointer assignments using a constraint graph. The analysis propagates points-to relations along the edges of the constraint graph, adding new edges as indirect constraints are resolved. Steensgaard's algorithm [23], which is an equality-based analysis, reduces the precision by using equality constraints and runs in almost linear time. That is, points-to information flows in bi-directions, rather than in uni-direction of the constraint. Compared with the related work, SEEKER achieves the same precision as that of Andersen's algorithm because we infer the points-to relations along the constraint edges whose propagation semantics is same as that of the Andersen's algorithm. Furthermore, SEEKER explores the points-to analysis using CFL reachability, which only computes the points-to relations for the variables of interest.

Based on the classical algorithms, researchers have developed a number of points-to analysis algorithms, which can be further characterized with respect to their context/flow sensitivities. Flow-sensitive and context-sensitive algorithms (FSCS) [6, 7, 25] are usually precise, but are not easy to scale to large programs. Kahlon [11] has proposed a bootstrapping analysis algorithm that partitions a program into small subsets and uses a divide-and-conquer strategy to concurrently find solutions in these subsets. He adopts a summary-based approach to make the context-sensitive alias analysis scalable. Yu et al. [29] has proposed a level-by-level algorithm that improves the scalability of the FSCS algorithm. Flow-insensitive and context-insensitive (FICI) algorithms [1, 23] have the best scalability on the large programs with overly conservative results. How well the algorithms scale to large programs is an important issue. Trade-offs between efficiency and precision are made by various points-to analyses, including flow-insensitive and context-sensitive (FICS) analyses [14, 17, 26, 31] and flow-sensitive and context-insensitive (FSCI) analyses [4, 9, 12, 15]. The related work adopts a uniform analysis algorithm to analyze the points-to relations for all the variables, while SEEKER enables a flexible mechanism of algorithm combinations to tradeoff the analysis cost and precision. Different points-to analysis and alias checking algorithms can be easily combined in SEEKER in order to achieve different precisions.

Demand-Driven Points-to Analysis. Demand-driven points-to analysis is a quick technique for obtaining the points-to sets of some variables. Reps et al. [18, 19] have led the research on program analysis via graph reachability. A number of static analysis problems were formulated in terms of CFL reachability, which boosts the study on demand-driven points-to analysis algorithms. A deduction-based demand-driven points-to analysis for C has been introduced by Heintze and Tardieu [10]. The method has the ability to compute the points-to sets of the queries on demand. Similarly, a demand-driven alias analysis for C has been introduced by Zheng and Rugina [30]. They have used a memory alias CFL reachability formulation, with indirect function calls which are conservatively handled in a context-insensitive way, thus reducing the precision for some queries. Sridharan and Bodík [21] have presented a CFL reachability formulation to model instance field accesses as a balanced-parentheses problem regardless of context-sensitivity. After that, they have extended their former work to achieve a context-sensitive points-to analysis [22] through adding the context-sensitivity check. Lu et al. [16] have proposed an incremental approach to demand-driven context-sensitive points-to analysis based on CFL reachability, which only recomputes the points-to sets affected by the program changes. The related work of demand-driven points-to analysis performs the computation on the flow-insensitive program representation not distinguishing the variables at the different program points. In

order to improve the precision of demand-driven analysis, SEEKER performs the points-to paths searching in a PtDG, which captures the points-to set propagations for the variables of different program points.

Some optimizations are applied to the demand-driven points-to analysis. Shang et al. [20] have improved the performance of demand-driven context-sensitive points-to analysis through adopting a method-level partial points-to analysis as the pre-analysis. Xu et al. [27] have proposed another approach *memAlias* to improve the performance of the refinement-based context-sensitive points-to analysis [21] that builds a symbolic graph to reduce the PAG size of a program. Yan et al. [28] have proposed a demand-driven alias analysis which extends *memAlias*. Nevertheless, it checks the alias without computing the points-to sets of the variables. SEEKER improves these techniques by modeling the control flow information in the CFL definition, and thus it reduces the pairs of matching between load and store statements avoiding the computation for the invalid paths of the CFG.

7. Conclusions

SEEKER is a flow-sensitive approach to demand-driven points-to analysis of Java program variables. It performs a query of the points-to set of a variable by constructing a PtDG with strong updates in the linear time, and adopting the context-free language (CFL) reachability to perform a quick analysis of the points-to sets of all variables the objective variable depends on. We have designed two generic algorithms to support the demand-driven points-to analysis via CFL reachability and six implementations of the generic algorithms which combine different context-sensitive models and alias checking algorithms.

In the future, we would like to parallelize the points-to set queries to improve the performance of SEEKER. An extension of SEEKER to an incremental points-to analysis is also part of our future work.

Acknowledgements

This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No.60970009 and 60673120).

References

- [1] L. Andersen. Program analysis and specialization for the C programming language. DIKU report 94-19, University of Copenhagen, 1994.
- [2] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*, pages 332–341, 2005.
- [3] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Transactions on Software Engineering*, 27(6):481–512, 2001.
- [4] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, 1993.
- [5] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Static Analysis Symposium*, pages 260–278, 2001.
- [6] A. De and D. D’Souza. Scalable flow-sensitive pointer analysis for Java with strong updates. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, pages 665–687, 2012.
- [7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
- [8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.
- [9] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, 2009.
- [10] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.
- [11] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the ACM SIGPLAN 2008 conference on Programming Language Design and Implementation*, pages 249–259, 2008.
- [12] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, 2011.
- [13] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, 2003.
- [14] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction*, pages 47–64, 2006.
- [15] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and the 13rd European Software Engineering Conference*, pages 343–353, 2011.
- [16] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*, pages 61–81, 2013.
- [17] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans-*

- actions on Software Engineering and Methodology, 14:1–41, 2002.
- [18] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.
- [19] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.
- [20] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 264–274, 2012.
- [21] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 2006.
- [22] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, 2005.
- [23] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- [24] D. Vanoverberghe, N. Tillmann, and F. Piessens. Test input generation for programs with pointers. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 277–291, 2009.
- [25] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, 2002.
- [26] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, 2004.
- [27] G. H. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23th European Conference on Object-Oriented Programming*, pages 98–122, 2009.
- [28] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, pages 155–165, 2011.
- [29] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization*, pages 218–229, 2010.
- [30] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, 2008.
- [31] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145–157, 2004.